

---

# Templates

---

## Key Concepts

Generic programming | Multiple parameters in class templates | Function templates | Template functions | Member function templates | Class templates | Template classes | Multiple parameters in class templates | Overloading of template functions | Nontype template arguments

### 12.1

## Introduction

*Templates* is one of the features added to C++ recently. It is a new concept which enable us to define generic classes and functions and thus provides support for *generic programming*. Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.

A template can be used to create a family of classes or functions. For example, a class template for an **array** class would enable us to create arrays of various data types such as **int** array and **float** array. Similarly, we can define a template for a function, say **mul()**, that would help us create various versions of **mul()** for multiplying **int**, **float** and **double** type values.

A template can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type. Since a template is defined with a *parameter* that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called *parameterized classes or functions*.

### 12.2

## Class Templates

Consider a vector class defined as follows:

```
class vector
{
    int *v;
    int size;
public:
    vector(int m)           // create a null vector
    {
        v = new int[size = m];

        for(int i=0; i<size; i++)
            v[i] = 0;
    }
    vector(int *a)           // create a vector from an array
    {
        for(int i=0; i<size; i++)
            v[i] = a[i];
    }
    int operator*(vector &y)   // scalar product
    {
        int sum = 0;
        for(int i=0; i<size; i++)
            sum += this->v[i] * y.v[i];
        return sum;
    }
};
```

The vector class can store an array of **int** numbers and perform the scalar product of two **int** vectors as shown below:

```
int main()
{
    int x[3] = {1,2,3};
    int y[3] = {4,5,6};
    vector v1(3);           // Creates a null vector of 3 integers
    vector v2(3);
```

```

v1 = x;                // Creates v1 from the array x
v2 = y;
int R = v1 * v2;
cout << "R = " << R;
return 0;
}

```

Now, suppose we want to define a vector that can store an array of **float** values. We can do this by simply replacing the appropriate **int** declarations with **float** in the **vector** class. This means that we have to **redefine the entire class all over again**.

Assume that we want to define a **vector** class with the data type as a parameter and then use this class to create a vector of **any data type** instead of **defining a new class every time**. The template mechanism enables us to **achieve this goal**.

As mentioned earlier, templates allow us to define generic classes. It is a simple process to create a generic class using a template with an **anonymous type**. The **general format** of a class template is:

```

template<class T>
class classname
{
    // .....
    // class member specification
    // with anonymous type T
    // wherever appropriate
    // .....
};

```

The template definition of **vector** class shown below illustrates the syntax of a template:

```

template<class T>
class vector

```

```

{
    T* v;      // Type T vector
    int size;
public:
    vector(int m)
    {
        v = new T[size = m];
        for(int i=0; i<size; i++)
            v[i] = 0;
    }
    vector(T* a)
    {
        for(int i=0; i<size; i++)
            v[i] = a[i];
    }
    T operator*(vector &y)
    {
        T sum = 0;
        for(int i=0; i<size; i++)
            sum += this -> v[i] * y . v[i];
        return sum;
    }
};

```



**NOTE:** The class template definition is very similar to an ordinary class definition except the prefix **template<class T>** and the use of type **T**. This prefix tells the compiler that we are going to declare a template and use **T** as a type name in the declaration. Thus, vector has become a parameterized class with the type **T** as its parameter. **T** may be substituted by any data type including the user-defined types. Now, we can create vectors for holding different data types.

Example:

```
vector <int> v1(10);      // 10 element int vector
vector <float> v2(25);    // 25 element float vector
```



**NOTE:** The type *T* may represent a class name as well.  
Example:

```
vector <complex> v3(5);    // vector of 5 complex numbers
```

A class created from a class template is called a *template class*. The **syntax** for defining an object of a **template class** is:

```
classname<type> objectname(arglist);
```

This process of creating a specific class from a class template is called *instantiation*. The compiler will perform the error analysis only when an instantiation takes place. It is, therefore, advisable to create and debug an ordinary class before converting it into a template.

Programs 12.1 and 12.2 illustrate the use of a **vector** class template for performing the scalar product of **int** type vectors as well as **float** type vectors.

## Program 12.1 Example of Class Template

```
#include <iostream>

using namespace std;

const size = 3;
```

```

template <class T>
class vector
{
    T* v;          // type T vector
public:
    vector()
    {
        v = new T[size];
        for(int i=0;i<size;i++)
            v[i] = 0;
    }
    vector(T* a)
    {
        for(int i=0;i<size;i++)
            v[i] = a[i];
    }
    T operator*(vector &y)
    {
        T sum = 0;
        for(int i=0;i<size;i++)
            sum += this -> v[i] * y.v[i];
        return sum;
    }
    void display(void)
    {
        for(int i=0;i<size;i++)
            cout<<v[i]<<"\t";
        cout<<"\n";
    }
};

int main()
{
    int x[3] = {1,2,3};
    int y[3] = {4,5,6};
    vector <int> v1;
    vector <int> v2;
    v1 = x;

```

```

        v2 = y;

        cout<<"v1 = ";
        v1.display();

        cout<<"v2 = ";
        v2.display();

        cout<<"v1 X v2 = "<<v1*v2;

        return 0;
    }

```

The output of the Program 12.1 would be:

```

v1 = 1  2  3
v2 = 4  5  6
v1 X v2 = 32

```

## Program 12.2 Another Example of Class Template

```

#include <iostream>

using namespace std;

const size = 3;
template <class T>
class vector
{
    T* v;           // type T vector
public:
    vector()

```



```

{
    v = new T[size];
    for(int i=0;i<size;i++)
        v[i] = 0;
}
vector(T* a)
{
    for(int i=0;i<size;i++)
        v[i] = a[i];
}
T operator*(vector &y)
{
    T sum = 0;
    for(int i=0;i<size;i++)
        sum += this -> v[i] * y.v[i];
    return sum;
}

void display(void)
{
    for(int i=0;i<size;i++)
        cout<<v[i]<<"\t";
    cout<<"\n";
}
};
int main()
{
    float x[3] = {1.1,2.2,3.3};
    float y[3] = {4.4,5.5,6.6};
    vector <float> v1;
    vector <float> v2;
    v1 = x;
    v2 = y;

    cout<<"v1 = ";
    v1.display();
}

```

```

        cout<<"v2 = ";
        v2.display();

        cout<<"v1 X v2 = "<<v1*v2;

        return 0;
    }

```

The output of the Program 12.2 would be:

```

v1 = 1.1    2.2    3.3
v2 = 4.4    5.5    6.6

v1 X v2 = 38.720001

```

## 12.3 Class Templates with Multiple Parameters

We can use more than one generic data type in a class template. They are declared as a comma-separated list within the **template** specification as shown below:

```

template<class T1, class T2, ...>
class classname
{
    .....
    ..... (Body of the class)
    .....
};

```

Program 12.3 demonstrates the use of a template class with two generic data types.

## Program 12.3 Two Generic Data Types in a Class Definition

```
#include <iostream>

using namespace std;

template<class T1, class T2>
class Test
{
    T1 a;
    T2 b;
public:
    Test(T1 x, T2 y)
    {
        a = x;
        b = y;
    }
    void show()
    {
        cout << a << " and " << b << "\n";
    }
};

int main()
{
    cout<<"Instantiating the class template as test1 with float
        and int data types..\ntest1: ";
    Test<float,int>test1(1.23,123);
    test1.show();

    cout<<"Instantiating the class template as test2 with int
        and char data types..\ntest2: ";
    Test<int,char>test2(100,'W');
    test2.show();
}
```

```
        return 0;  
    }
```

The output of Program 12.3 would be:

Instantiating the class template as test1 with float and int data types..

test1: 1.23 and 123

Instantiating the class template as test2 with int and char data types..

test2: 100 and W

Just as we specify default values for function arguments, we may also specify default types for the template class data types. Program 12.4 modifies Program 12.3 to demonstrate default type specification:

## Program 12.4 Using Default Data Types in a Class Definition

```
#include <iostream.h>  
  
using namespace std;  
  
template<class T1=int, class T2=int> //default data types  
specified  
                                as int  
class Test  
{  
    T1 a;  
    T2 b;  
public: Test(T1 x, T2 y)
```

```

    {
        a = x;
        b = y;
    }
    void show()
    {
        cout<<a<<" and "<<b<<"\n";
    }
};
int main()
{
    Test <float,int> test1(1.23,123);
    Test <int, char> test2 (100, 'W');
    Test <int> test3 ('a', 12.983); //declaration of class object
    without types specification
    test1.show();
    test2.show();
    test3.show();
    return 0;
}

```

The output of Program 12.4 would be:

```

1.23 and 123
100 and W
97 and 12

```

The above program declares test3 object without any type specification, thus the default data type for T1 and T2 is considered as int. The parameter values passed by test3 are type casted to int and displayed as output.

## 12.4

## Function Templates

Like class templates, we can also define function templates that could be used to create a family of functions with different argument types. The general format of a function template is:

```
template<class T>
returntype functionname (arguments of type T)
{
    // .....
    // Body of function
    // with type T
    // wherever appropriate
    // .....
}
```

The function template syntax is similar to that of the class template except that we are defining functions instead of classes. We must use the template parameter **T** as and when necessary in the function body and in its argument list.

The following example declares a **swap()** function template that will swap two values of a given type of data.

```
template<class T>
void swap(T&x, T&y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

This essentially declares a set of overloaded functions, one for each type of data. We can invoke the **swap()** function like any ordinary function. For example, we can apply the **swap()** function as follows:

```

void f(int m,int n,float a,float b)
{
    swap(m,n); // swap two integer values
    swap(a,b); // swap two float values
    // .....

```

This will generate a **swap()** function from the function template for each set of argument types. Program 12.5 shows how a template function is defined and implemented.

## Program 12.5 Function Template - An Example

```

#include <iostream>

using namespace std;

template <class T>
void swap(T &x, T &y)
{
    T temp = x;
    x = y;
    y = temp;
}

void fun(int m,int n,float a,float b)
{
    cout << "m and n before swap: " << m << " " << n << "\n";
    swap(m,n);
    cout << "m and n after swap: " << m << " " << n << "\n";

    cout << "a and b before swap: " << a << " " << b << "\n";
    swap(a,b);
    cout << "a and b after swap: " << a << " " << b << "\n";
}

```

```

int main()
{
    fun(100,200,11.22,33.44);

    return 0;
}

```

The output of Program 12.5 would be:

```

m and n before swap:  100 200
m and n after swap:   200 100
a and b before swap:  11.22 33.439999
a and b after swap:   33.439999 11.22

```

Another function often used is **sort()** for sorting arrays of various types such as **int** and **double**. The following example shows a function template for bubble sort:

```

template<class T>
bubble(T v[], int n)
{
    for(int i=0; i<n-1; i++)
        for(int j=n-1; i<j; j--)
            if(v[j] < v[j-1])
            {
                T temp = v[j];
                v[j] = v[j-1];
                v[j-1] = temp;
            }
}

```

Note that the swapping statements

```

T temp = v[j];
v[j] = v[j-1];
v[j-1] = temp;

```



may be replaced by the statement

```
swap(v[j],v[j-1]);
```

where **swap()** has been defined as a function template.

Here is another example where a function returns a value.

```
template<class T>
T max(T x, T y)
{
    return x>y ? x:y;
}
```

A function generated from a function template is called a *template function*. Program 12.6 demonstrates the use of two template functions in nested form for implementing the bubble sort algorithm. Program 12.7 shows another example of application of template functions.

## Program 12.6 Bubble Sort Using Template Functions

```
#include <iostream>

using namespace std;

template<class T>
void bubble(T a[], int n)
{
    for(int i=0; i<n-1; i++)
        for(int j=n-1; i<j; j--)
            if(a[j] < a[j-1])
                {
```

```

        swap(a[j],a[j-1]);           // calls template function
    }
}
template<class X>
void swap(X &a, X &b)
{
    X temp = a;
    a = b;
    b = temp;
}
int main()
{
    int x[5] = {10,50,30,40,20};
    float y[5] = {1.1,5.5,3.3,4.4,2.2};

    bubble(x,5); // calls template function for int values
    bubble(y,5); // calls template function for float values
    cout << "Sorted x-array: ";
    for(int i=0; i<5; i++)
        cout << x[i] << " ";
    cout << endl;

    cout << "Sorted y-array: ";
    for(int j=0; j<5; j++)
        cout << y[j] << " ";
    cout << endl;

    return 0;
}

```

The output of Program 12.6 would be:

```

Sorted x-array: 10 20 30 40 50
Sorted y-array: 1.1 2.2 3.3 4.4 5.5

```

## Program 12.7 An Application of Template Function

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

template <class T>
void roots(T a,T b,T c)
{
    T d = b*b - 4*a*c;
    if(d == 0)                                // Roots are equal
    {
        cout << "R1 = R2 = " << -b/(2*a) << endl;
    }
    else if(d > 0)                            // Two real roots
    {
        cout << "Roots are real \n";
        float R = sqrt(d);
        float R1 = (-b+R)/(2*a);
        float R2 = (-b-R)/(2*a);
        cout << "R1 = " << R1 << " and ";
        cout << "R2 = " << R2 << endl;
    }
    else                                      // Roots are complex
    {
        cout << "Roots are complex \n";
        float R1 = -b/(2*a);
        float R2 = sqrt(-d)/(2*a);
        cout << "Real part = " << R1 << endl;
        cout << "Imaginary part = " << R2;
        cout << endl;
    }
}
```

```

    }
}
int main()
{
    cout << "Integer coefficients \n";
    roots(1,-5,6);
    cout << "\nFloat coefficients \n";
    roots(1.5,3.6,5.0);

    return 0;
}

```

The output of Program 12.7 would be:

Integer coefficients  
Roots are real

R1 = 3 and R2 = 2

Float coefficients  
Roots are complex  
Real part = -1.2  
Imaginary part = 1.375985

## 12.5 Function Templates with Multiple Parameters

Like template classes, we can use more than one generic data type in the template statement, using a comma-separated list as shown below:

```

template<class T1, class T2, ...>
    returntype functionname (arguments of types T1, T2,...)
    {
        .....
    }

```

```
..... (Body of function)
.....
}
```

Program 12.8 illustrates the concept of using two generic types in template functions.

## Program 12.8 Function with Two Generic Types

```
#include <iostream>
#include <string>
using namespace std;

template<class T1, class T2>
void display(T1 x, T2 y)
{
    cout << x << " " << y << "\n";
}

int main()
{
    cout<<"Calling function template with int and character
    string type parameters...\n";
    display(1999, "EBG");
    cout<<"Calling function template with float and integer type
    parameters...\n";
    display(12.34, 1234);

    return 0;
}
```

The output of Program 12.8 would be:

Calling function template with int and character string type parameters...

1999 EBG

Calling function template with float and integer type parameters...

12.34 1234

## 12.6 Overloading of Template Functions

A template function may be overloaded either by template functions or ordinary functions of its name. In such cases, the overloading resolution is accomplished as follows:

1. Call an ordinary function that has an exact match.
2. Call a template function that could be created with an exact match.
3. Try normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match is found. Note that no automatic conversions are applied to arguments on the template functions. Program 12.9 shows how a template function is overloaded with an explicit function.

### Program 12.9 Template Function With Explicit Function

```
#include <iostream.h>
#include <string.h>

using namespace std;
```

```

template <class T>
void display(T x)          //overloaded template function display
{
    cout<<"Overloaded Template Display 1: "<<x<<"\n";
}
template <class T, class T1>
void display(T x, T1 y)    //overloaded template function
display
{
    cout<<"Overloaded Template Display 2: "<<x<<", "
<<y<<"\n";
}
void display(int x)        //overloaded generic display function
{
    cout<<"Explicit display: "<<x<<"\n";
}
int main()
{
    display(100);
    display(12.34);
    display(100,12.34);
    display('C');

    return 0;
}

```

The output of Program 12.9 would be:

Explicit display: 100

Overloaded Template Display 1: 12.34

Overloaded Template Display 2: 100, 12.34

Overloaded Template Display 1: C



**NOTE:** The call ***display(100)*** invokes the ordinary version of ***display()*** and not the template version.

## 12.7 Member Function Templates

When we created a class template for vector, all the member functions were defined as inline which was not necessary. We could have defined them outside the class as well. But remember that the member functions of the template classes themselves are parameterized by the type argument (to their template classes) and therefore these functions must be defined by the function templates. It takes the following general form:

```
Template<class T>
returntype classname <T> :: functionname(arglist)
{
    // .....
    // Function body
    // .....
}
```

The **vector** class template and its member functions are redefined as follows:

```
// Class template .....
```

```
template<class T>
class vector
{
    T* v;
    int size;
public:
```



```

        vector(int m);
        vector(T* a);
        T operator*(vector & y);
};
// Member function templates .....

template<class T>
vector<T> :: vector(int m)
{
    v = new T[size = m];
    for(int i=0; i<size; i++)
        v[i] = 0;
}
template< class T>
vector<T> :: vector(T* a)
{
    for(int i=0; i<size; i++)
        v[i] = a[i];
}
template< class T>
T vector<T> :: operator*(vector & y)
{
    T sum = 0;
    for(int i = 0; i < size; i++)
        sum += this -> v[i] * y.v[i];
    return sum;
}

```

## 12.8 Nontype Template Arguments

We have seen that a template can have multiple arguments. It is also possible to use non-type arguments. That is, in addition to the type argument **T**, we can also use other arguments such as strings, function names, constant expressions and built-in types. Consider the following example:

```

template<class T, int size>
class array
{
    T a[size];                // automatic array initialization
                              // .....
                              // .....
};

```

This template supplies the size of the array as an argument. This implies that the size of the array is known to the compiler at the compile time itself. The arguments must be specified whenever a template class is created. Example:

```

array<int,10> a1;           // Array of 10 integers
array<float,5> a2;         // Array of 5 floats
array<char,20> a3;         // String of size 20

```

The size is given as an argument to the template class.

## Summary

- ☐ C++ supports a mechanism known as template to implement the concept of generic programming.
- ☐ Templates allows us to generate a family of classes or a family of functions to handle different data types.
- ☐ Template classes and functions eliminate code duplication for different types and thus make the program development easier and more manageable.
- ☐ We can use multiple parameters in both the class templates and function templates.

- ☐ A specific class created from a class template is called a template class and the process of creating a template class is known as instantiation. Similarly, a specific function created from a function template is called a template function.
- ☐ Like other functions, template functions can be overloaded.
- ☐ Member functions of a class template must be defined as function templates using the parameters of the class template.
- ☐ We may also use nontype parameters such basic or derived data types as arguments templates.

## Key Terms

bubble sort | class template | **display()** | **explicit** function | function template | generic programming | instantiation | member function template | multiple parameters | overloading | **parameter** | parameterized classes | parameterized functions | swapping | **swap()** | **template** | template class | template definition | template function | template parameter | template specification | templates

## Review Questions

---

- 12.1 What is generic programming? How is it implemented in C++?
- 12.2 Explain with the help of an example why templates are used in programming?
- 12.3 A template can be considered as a kind of macro. Then, what is the difference between them?
- 12.4 Distinguish between overloaded functions and function templates.

**12.5** Distinguish between the terms class template and template class.

**12.6** A class (or function) template is known as a parameterized class (or function). Comment.

**12.7** State which of the following definitions are illegal.

(a) `template<class T>`

```
class city  
{.....};
```

(b) `template<class P, R, class S>`

```
class city  
{.....}
```

(c) `template<class T, typename S>`

```
class city  
{.....};
```

(d) `template<class T, typename S>`

```
class city  
{.....};
```

(e) `class<class T, int size=10>`

```
class list  
{.....};
```

(f) `class<class T = int, int size>`

```
class list  
{.....};
```

**12.8** Identify which of the following function template definitions are illegal.

**(a)** `template<class A, B>`

```
void fun(A, B)
{.....};
```

**(b)** `template<class A, class A>`

```
void fun(A, A)
{.....};
```

**(c)** `template<class A>`

```
void fun(A, A)
{.....};
```

**(d)** `template<class T, typename R>`

```
T fun(T, R)
{.....};
```

**(e)** `template<class A>`

```
A fun(int *A)
{.....};
```

## Debugging Exercises

---

**12.1** Identify the error in the following program.

```
#include <iostream.h>
class Test
{
    int intNumber;
    float floatNumber;
public:
    Test()
```

```

    {
        intNumber = 0;
        floatNumber = 0.0;
    }

    int getNumber()
    {
        return intNumber;
    }

    float getNumber()
    {
        return floatNumber;
    }
};

void main()
{
    Test objTest1;
    objTest1.getNumber();
}

```

**12.2** Identify the error in the following program.

```

#include <iostream.h>

template <class T1, class T2>
class Person
{
    T1 m_t1;
    T2 m_t2;
public:
    Person (T1 t1, T2 t2)
    {
        m_t1 = t1;
        m_t2 = t2;
        cout << m_t1 << " " << m_t2 << endl;
    }
}

```

```

    }

    Person (T2 t2, T1 t1)
    {
        m_t2 = t2;
        m_t1 = t1;
        cout << m_t1 << " " << m_t2 << endl;
    }
};

void main()
{
    Person<int, float> objPerson1(1, 2.345);
    Person<float, char> objPerson2(2.132, 'r');
}

```

**12.3** Identify the error in the following program.

```

#include <iostream.h>

template <class T1, class T2>
T1& MinMax(T1 t1, T2 t2)
{
    return t1 > t2 ? t1 : t2;
    cout << " ";
}

void main()
{
    cout << ++MinMax(2, 3);
}

```

**12.4** Find errors, if any, in the following code segment.

```

template<class T>
T max(T, T)
{.....};
unsigned int m;
int main()
{

```

```
        max(m, 100);  
    }
```




**12.5** Identify the error in the following program:

```
#include <iostream.h>  
  
using namespace std;  
  
template<class T1=int, class T2>  
class Test  
{  
    T1 a;  
    T2 b;  
public:  
    Test(T1 x, T2 y)  
    {  
        a=x;  
        b=y;  
    }  
    void show()  
    {  
        cout<<a<<" and "<<b<<"\n";  
    }  
};  
  
int main()  
{  
    Test <float,int> test1(1.23,123);  
    Test <float> test2 (9.99,12.983);  
    test1.show();  
    test2.show();  
  
    return 0;  
}
```



## Programming Exercises

---

- 12.1** Write a function template for finding the minimum value contained in an array. .
- 12.2** Write a class template to represent a generic vector. Include member functions to perform the following tasks:
- (a) To create the vector.
  - (b) To modify the value of a given element.
  - (c) To multiply by a scalar value.
  - (d) To display the vector in the form (10,20,30.....). .
- 12.3** Write a function template to perform linear search in an array. .